2-1-2012

Improving security and usability of mobile device authentication mechanisms

Michael Pinch

Improving Security and Usability of Mobile Device Authentication Mechanisms

Thesis

By Michael Pinch

Master of Science in Computer Security and Information Assurance

Networking, Security, and System Administration

Golisano College of Computing and Information Sciences

Professor Charles Border, RIT

Professor Yin Pan, RIT

Professor Joe O'Donnell, Canisius College

Rochester Institute of Technology

Rochester, NY

February, 2012

Signature Page

_____

Professor Charles Border, RIT

_____

Professor Yin Pan, RIT

_____

Professor Joe O'Donnell, Canisius College

Abstract

Mobile devices, in the form of smartphones and tablet computers, are going through an amazing growth cycle. The devices are powerful and robust enough to house an incredible amount of sensitive data about our personal and business lives. These devices, however, have relatively weak authentication systems, generally consisting of a pin number or pattern matching challenge. In addition to the weak authentication systems, the input mechanism of using a finger on a touchscreen leaves a residue trail that can be easily recovered, allowing an attacker to recover some or all of the authentication code. This thesis examines the strengths and weaknesses of the authentication systems available on iOS and Android systems. It then looks for alternative improved solutions by examining the array of sensor technologies on current mobile devices. Three improved solutions are presented, including a dynamic touchscreen pattern that removes the ability for a smudge attack, a forced rotation authentication screen that obfuscates input patterns, and a GPS enabled system that reduces authentication requirements when in a designated "safe zone".

Table of Contents

Introduction

Background

The smartphone market has seen explosive growth since the introduction of the Apple iPhone in 2007. A number of viable competitors have emerged since, and the smartphone is quickly becoming ubiquitous amongst nearly all market segments, spanning all age groups as well as business / personal users. A similar explosion, while earlier in the adoption curve, is being seen in the tablet computer market with the release of the Apple iPad and various Android tablets. These devices have the computing power, sensors, and supporting application availability to allow them to house just as much, if not more, data than laptops and desktops. With the criticality of information being stored on these devices, security is becoming an important area of focus. For example, the NSA recently released a highly secured version of Android, based on SELinux, called SEAndroid. (Hoover, 2012) This version of Android goes to great length to protect the runtime space of each application, to protect critical applications from potentially malicious applications installed on the device. One critical area that is often overlooked is the first layer of defense; the unlock screen. The unlock screen is corollary in most cases to an operating system password on traditional computers, an area that has seen much attention throughout the years to find ways to improve security. The available lock screen authentication challenges have weaknesses in terms of overall complexity as well as by their input method. Lock screen authentication challenges require a user to

unlock the device by interacting with a touch screen. This interaction leaves significant residue on the screen that can allow an attacker to reverse engineer the authentication code. This thesis examines the growing smartphone market and the currently available authentication challenges that are in use. It examines their overall complexity and points out weaknesses in their design. Finally, it examines the available technologies common on current smartphones to identify alternative and improved methods for authentication that are not subject to the previously identified weaknesses.

Growing market

Since the introduction of the iPhone by Apple Computer in 2007, smart phones have gone from niche specialty items to devices carried by nearly 50% of the American population. Smartphones have expanded to offer similar to features to nearly all of those of the desktop computer, and due to their mobility and advanced sensing devices, offer a number of abilities that cannot be performed with a common personal computer. Smartphones make up 40% (Nielsen Communications, 2011) of the current market for mobile phones. An overall breakdown can be seen in illustration 1 below.

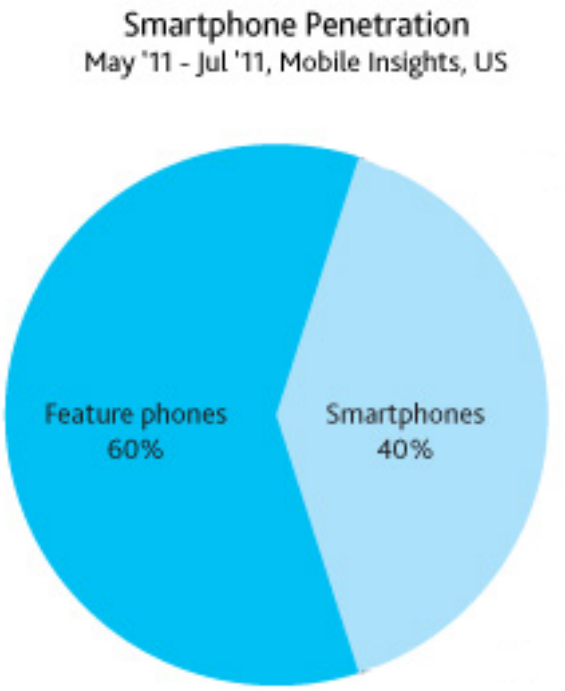


Illustration 1 (Nielsen Communications, 2011)

Even more compelling, 62% of the 25-34 year old market own and use smartphones (Nielsen, 2011), showing that the greatest saturation is in the age group that will continue to be targeted consumers for the next several decades.

The market for smartphones, in the past year has become largely a two-company race, between Android and iOS. While RIM (maker of Blackberry) still holds a sizeable market share at 19%, their trending shows them quickly moving into irrelevancy, losing 5% of their market share to iOS and Android in only a single quarter of 2011. (comScore, 2011) Illustration 2 shows a breakdown by operating system the market share of smartphone devices.

Top Smartphone Platforms 3 Month Avg. Ending Aug. 2011 vs. 3 Month Avg. Ending May 2011 Total U.S. Smartphone Subscribers Ages 13+ Source: comScore MobiLens			
	Share (%) of Smartphone Subscribers		
	May-11	Aug-11	Point Change
Total Smartphone Subscribers	*100.0%*	*100.0%*	*N/A*
Google	38.1%	43.7%	5.6
Apple	26.6%	27.3%	0.7
RIM	24.7%	19.7%	-5.0
Microsoft	5.8%	5.7%	-0.1
Symbian	2.1%	1.8%	-0.3

Illustration 2 (comScore, 2011)

In addition to the smartphone space, in 2010, the tablet computer device finally became a regularly accepted form factor. The release of the Apple iPad, soon followed by a variety of Android tablets, almost immediately made the tablet market a very similar 2-platform race. A study released by Gartner in 2011 shows the market being largely iOS dominated, but with the gap quickly closing with Android, as shown in Illustration 3. (Perez, 2011)

Table 1
Worldwide Sales of Media Tablets to End Users by OS (Thousands of Units)

OS	2010	2011	2012	2015
Android	2,512	11,020	22,875	116,444
iOS	14,685	46,697	69,025	148,674
MeeGo	179	476	490	197
Microsoft	0	0	4,348	34,435
QNX	0	3,016	6,274	26,123
WebOS	0	2,053	0	0
Other Operating Systems	235	375	467	431
Total Market	**17,610**	**63,637**	**103,479**	**326,304**

Source: Gartner (September 2011)

Illustration 3 (Perez, 2011)

While all the above listed smartphone tablet platforms share common authentication mechanisms, for the purpose of this analysis, the platforms of iOS and Android will be focused on due to their majority market share and the apparent trend towards market domination for these two platforms.

Smartphones and tablets today contain a wide array of sensors and technologies that make them incredibly powerful devices. Smartphone and tablet devices today have the ability to store our:

- Contacts
- Business and Personal Emails

- Personal Photos + Videos
- Usernames and Passwords
- Social Networking Identities
- Work Materials – Presentations, Documents, etc
- Banking Information

In the near future, there will be heavy adoption of even more advanced features, with smartphones function as:

- Car Keys
- Credit Card
- House Keys

As can be seen, a great deal of an individual's life can be stored and maintained on a mobile device. These devices have become critical for daily business, and a pervasive solution for consumers. Securing access to these devices should be considered absolutely critical, on both the business and consumer side of the market, due to their high degree of integration with critical aspects of the user's life.

Current Mobile Authentication Mechanisms

Current mobile phone authentication mechanisms on Android and iOS give few choices. They rely on memory-based pin-numbers, passwords, or pattern matching,

taken as input through the touchscreen of the device. All systems provide the minimum level of security necessary to keep intruders at bay, however suffer from some weaknesses due to the method of entry.

iOS

Pin Number Authentication

The iOS system by default (when authentication is enabled) provides a four digit numeric pin-code challenge to the user attempting to login to the device. This system is generally on par with overall cryptographic complexity to using an automated teller machine (ATM). See example 1 for an illustration of the interface presented to the user.

Example 1

Calculating the number of possible pin codes is a trivial task, as the user must have 4 digits, selecting from integers 0-9. The user may select any combination of numbers, with no limit on repetition. The user may not insert any blank characters, and therefore must have exactly 4 integers selected. For each authentication attempt, a user will have the following available password choices:

Input 1: 10 choices

Input 2: 10 choices

Input 3: 10 choices

Input 4: 10 choices

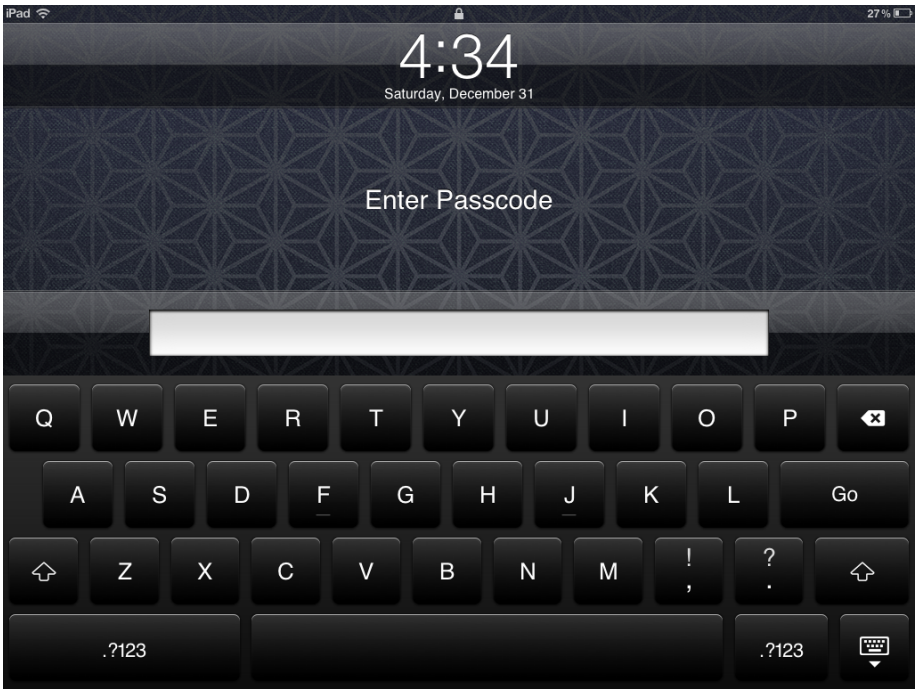
Therefore the number of possible authentication codes (AC) for the iOS pin system is:

$AC = 10 \times 10 \times 10 \times 10 = 10^4 = 10{,}000$

This particular system is widely used for iOS authentication both because it is the default security option, and it is trivial to input. This system is also successful due to the size and accessibility of the number pad, allowing users to navigate the authentication with one hand, or even without looking by remembering the basic tap pattern on the screen.

Password Authentication

The iOS system also allows for standard password authentication, utilizing the full available character set of the iOS keyboard. This type of authentication needs to be opted into above and beyond the standard pin code authentication. It is case sensitive, and has a minimum of 1 character length, and no observable maximum (up to 40 characters was tested). See example 1 for an illustration of the interface presented to the user.



Example 2

The available character set for authentication is:

26 lowercase letters

26 uppercase letters

10 integers

34 punctuation / space / special characters

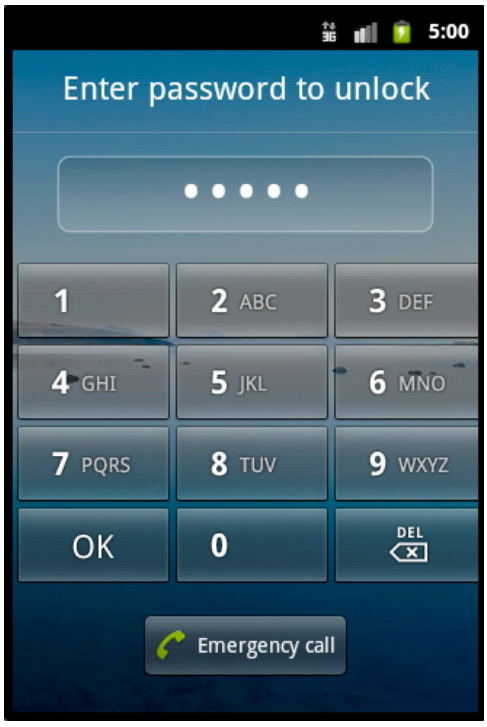
While the computation complexity of authentication via this mechanism can be quite high given the relatively large character set and lack of required password lengths, there are no requirements for complexity. Requirements such as utilization of an uppercase letter, a number, punctuation, or forced minimum length add significant complexity to passwords and prevent users from creating weak passwords. The absence of any of these requirements creates a situation in which there is likely possibility of a user creating a trivial password. The minimum number of available authentication codes (AC) can then be calculated as

$$AC = (26+26+10+34)^1 = 97$$

Android

Pin

The Android pin number system is very similar to the iOS pin number system. It allows the user a choice of 10 integers for an authentication code. This system differs from the iOS implementation of a pin number in that it does not limit the number of digits to use in the pin number. See example 3 for a view of the Android pin code authentication system.



Example 3

The Android system enforces a minimum of 4 digits, however a user may choose any number of digits to use for authentication. Effectively, this allows for increased security over the iOS mechanism as it provides an additional input choice after 4 characters have been entered. In other words, the selection of "OK" or submission of the pin code for authentication becomes a choice besides the standard available integers of 0-9. While the overall password complexity can be increased infinitely by the ability to extend the length of the pin number without limit, a minimum complexity can be obtained by viewing the following 4 choices:

Input 1: 10 choices (0-9)

Input 2: 10 choices (0-9)

Input 3: 10 choices (0-9)

Input 4: 10 choices (0-9)

Input 5: 11 choices (0-9 or "OK", indicating end of pin)

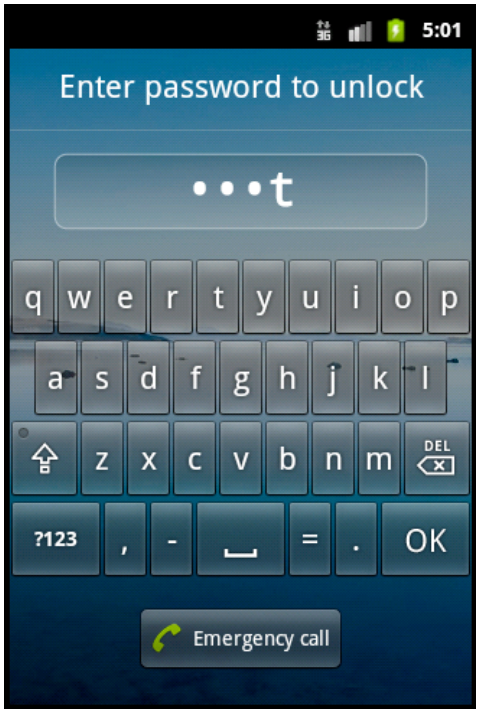
It can be seen that the Android approach of lifting the limit on maximum digits in a pin number creates a great increase in minimum complexity of passcodes over the IOS system. The minimum number of available authentication codes (AC) can then be calculated as

AC = 10x10x10x10x11 = 10^4 x 11 = 110,000

For pin numbers longer than 4 characters, the number of authentication codes (AC) increases by a factor of 10 with each additional digit added to the length of the pin.

Password

The Android password system differs little from the iOS system in that it allows a user to select a password of their choice from a wide range of characters. The only difference between the two systems is the forced 4 character minimum limit. The system enforces no complexity requirements besides minimum length. See example 4 for an illustration of the Android password authentication system:



Example 4

26 lowercase characters

26 uppercase characters

10 digits

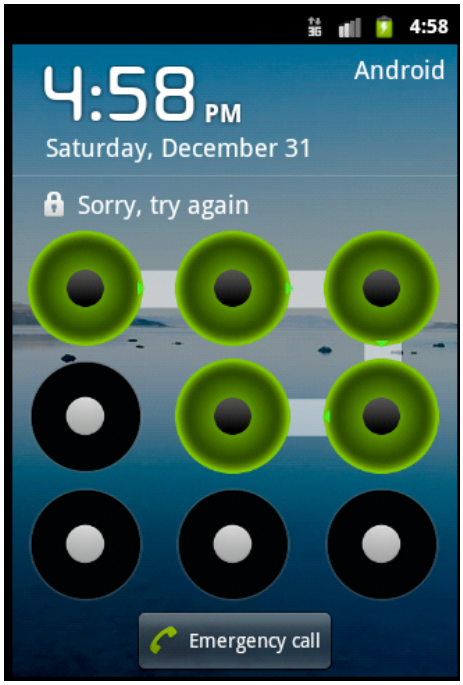
35 punctuation / space / special characters

As was noted with iOS password authentication, the large number of available characters and lack of a maximum character length allows for a huge amount of complexity. In this case the Android system once again has a slight advantage over the iOS system in terms of complexity as it has a slightly larger available character set, and it requires a minimum of 4 characters for the password system, which forces a minimum number of authentication codes (*AC*) of

$AC = (26 + 26 + 10 + 35)^4 = 97^4 = 88{,}529{,}281$

Pattern

One of the more popular and non-traditional available authentication systems is the Android standard pattern matching challenge. This pattern matching challenge requires that a user must swipe a finger over a set of nodes in the correct order. This type of system is of great convenience to the user, as it can typically be performed without a direct visual connection to the screen, as the pattern can be easily reproduced through muscle memory of the input finger. An example of this system can be seen in example 5.



Example 5

This pattern system follows a relatively small number of rules (Kaseorg, 2011)

- A valid pattern must have a minimum of 4 distinct nodes
- If the line connecting any two consecutive nodes in the pattern passes through any other nodes, those other nodes must have previously been in the pattern
- No node reuse is permitted

An object-oriented representation of each node's behavior was created to illustrate in detail the rules as they are enforced in the application, available in Appendix 1. This representation allows each instantiated node to report back whether it is a valid candidate, when supplied with a list of previously selected nodes.

The number of available patterns allowed under these rules is 389,112 (Aviv, Gibson, Mossop, Blaze, & Smith, 2010). An algorithm to perform a brute force calculation of the available patterns can be seen in Appendix 2 (Kaseorg, 2011). This places the overall complexity of the challenge greater than that of a 4-digit pin number, but less than that of the full password approach.

Issues with Existing Authentication Mechanisms

Hashing Algorithm

One potential downfall of the choice to make an operating system open source is that it exposes the underlying code to scrutiny. The Android operating system is itself open source. As such, the underlying mechanism for recording the lock screen pattern is available for review.

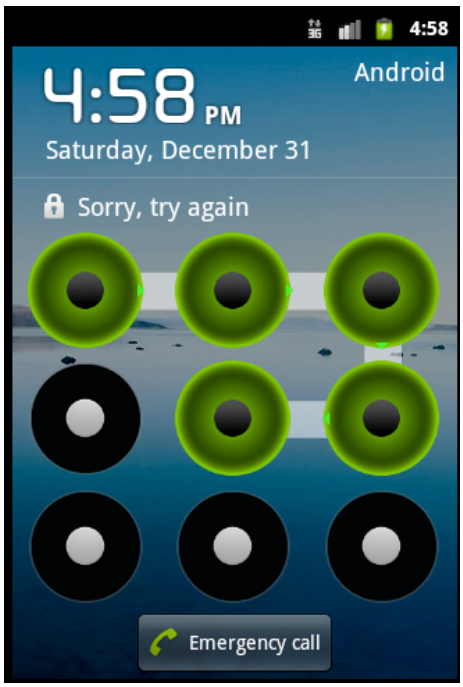
Contained within com.android.internal.widget.LockPatternUtils.java (Android Open Source Project) is the source code that manages the storage of the authentication pattern on the operating system. Through analysis of the code, it can be seen that the algorithm uses the NIST standard SHA-1 (secure hashing algorithm 1) hashing algorithm. The SHA-1 algorithm creates a 160 bit (40 character) hash to represent any set of input data (Secure Hash Standard, 1995). The implementation of the SHA-1 algorithm within the Android code can be seen within appendix 3, with specific focus on lines 458-483 (highlighted in appendix).

One side effect of this information being freely and publicly available is that it raises the risk of compromise. As was noted previously, the overall available key space for the Android pattern is 389,112. With such a relatively small key space, and a known and freely available hash and storage mechanism, a rainbow table for the entire key space can be generated with very little effort. The general Android file system is used for storage of the pattern hash. As such, without much effort, the pattern hash

can be recovered from a rooted Android device, and looked up in a hash table to recover the pattern.

Android assigns an integer value to each node, from left to right, then top to bottom, numbered sequentially from 0-8.

The example pattern contained below shows the pattern 0,1,2,5,4



The Android system hashes this string of character for its storage.

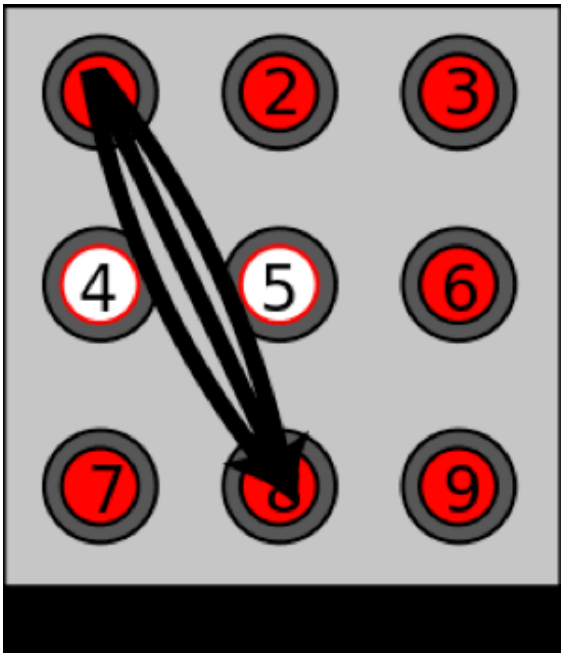
Sha1(0,1,2,5,4) = 061e48faa4a2971109dec3e7b20dd0ee1eab1c06

Based on this mechanism, a device may have its pattern recovered with relative ease by locating it on the device file system.

Smudge Attack

While a wide variety of what seem to be complex authentication mechanisms exist for mobile devices today, a critical issue hampers each of them. The method of input for these authentication systems is through a capacitive touch screen device. Capacitive touch screens calculate the input touch point by discharging a small current of electricity at the input point, meaning that to register a touch, the input must be through a conductive, electrically charged means, such as a human finger (Metro). As anyone who has ever used a touch screen device can quickly tell, a human finger leaves a large amount of residual oil behind on the screen after each touch. Due to the large number of times a user authenticates to a smart phone on a daily basis, this allows for a large amount of finger oil residue to build up on the points used repeatedly for authentication. This allows for what is referred to as the "Smudge Attack" (Aviv, Gibson, Mossop, Blaze, & Smith, 2010). This attack involves the inspection of residual smudges left on the device in order to reduce the number of possible authentication codes. Aviv et al showed that by minimal photographic analysis of a normally used smartphone, a 92% success rate for partial pattern recovery was possible, along with a 68% full recovery. For those devices on which only a partially recovered pattern was available, a high success rate for pattern authentication can be obtained by applying basic human use factors. For example, eliminating certain patterns that are technically possibly but inconvenient, immediately cuts the available number of patterns down to 158,410. Such a reduction, includes the 30° node pattern, which presents a high likelihood of

accidental triggering of non-intended nodes. In example 6 below, this type of attempt is illustrated as a pattern going from node 1 (upper left) to node 8, has a high likelihood of accidentally triggering nodes 4 and 5.



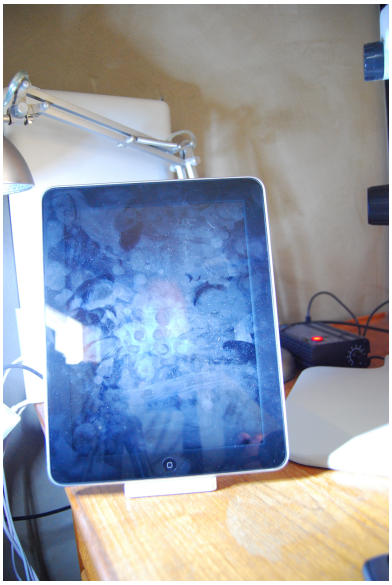
Example 6

Extending the Smudge Attack Logic to Pin Based Authentication

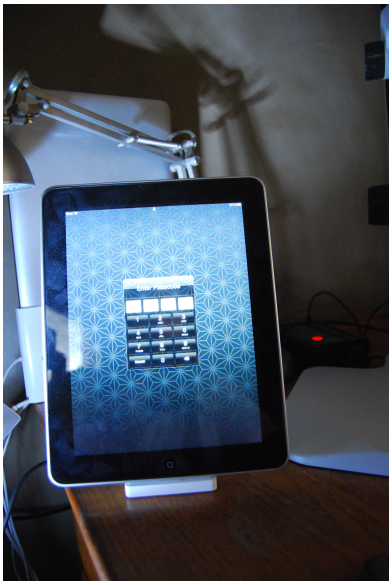
While the research done by Aviv, et al was focused on the Android based pattern matching authentication system, similar residual smudging is left behind when using pin number based authentication systems. In order to test whether a similar exploit was possible using a pin-based system, a similar test was run on a first generation iPad. The device was set configured to use a pin-number system, and

was used submitted to regular use for a day, without any dedicated cleaning or wiping of the device, but submitting it to incidental contact. After this period of use, the device was mounted on a stand, and photographed from a variety of angles and at a variety of apertures. The device was side-lit by a handheld LED microscope light in order to find the optimal angle for revealing residual smudges. The device, being photographed by a tripod-mounted camera, was also photographed turned on, awaiting pin-number input, to allow for an image overlay analysis to take place.

A side-by-side comparison of the two un-manipulated photos can be seen below in examples 7 and 8



Example 7



Example 8

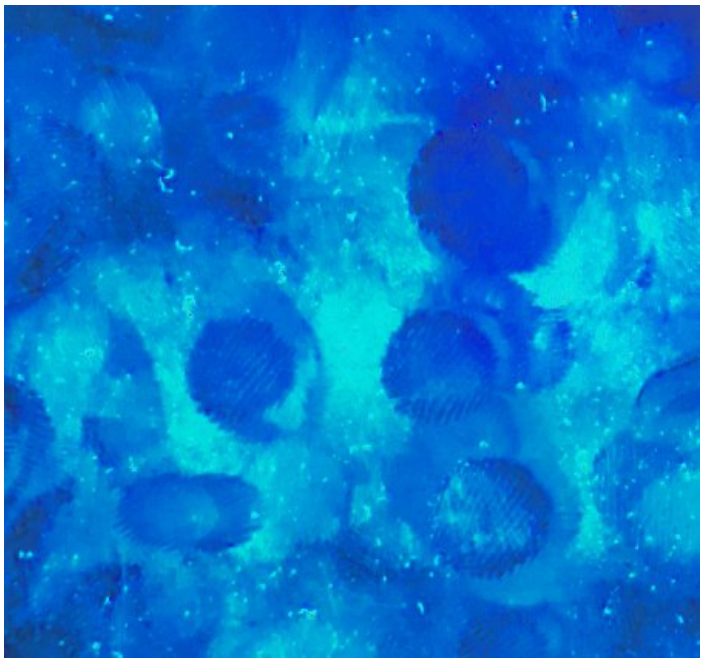
Utilizing the open source image processing software called Gimp, the following filters were applied to example 7:

Minimize lightness -100

Brightness +40

Contrast +106

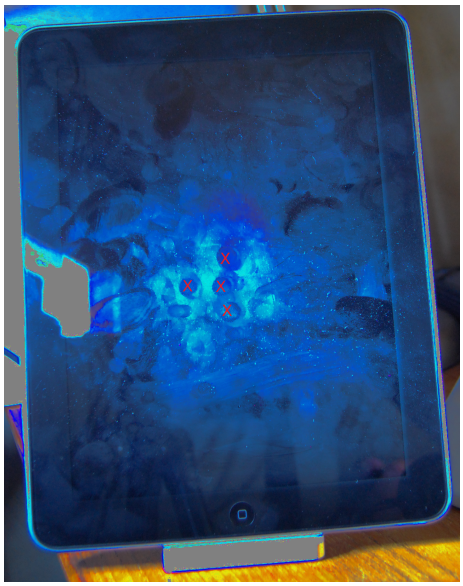
Applying these filters along with a close-up crop of the image allowed the extraction of the image in example 9.



Example 9

In analyzing examples 7 and 9, it became clear that the deliberate input of single pin numbers on a touch screen devices leaves a different signature than the typical use patterns of dedicated applications, which often rely on finger swipes. The deliberate pin number tapping process left clear residual fingerprints, with grooves and patterns clearly distinguishable.

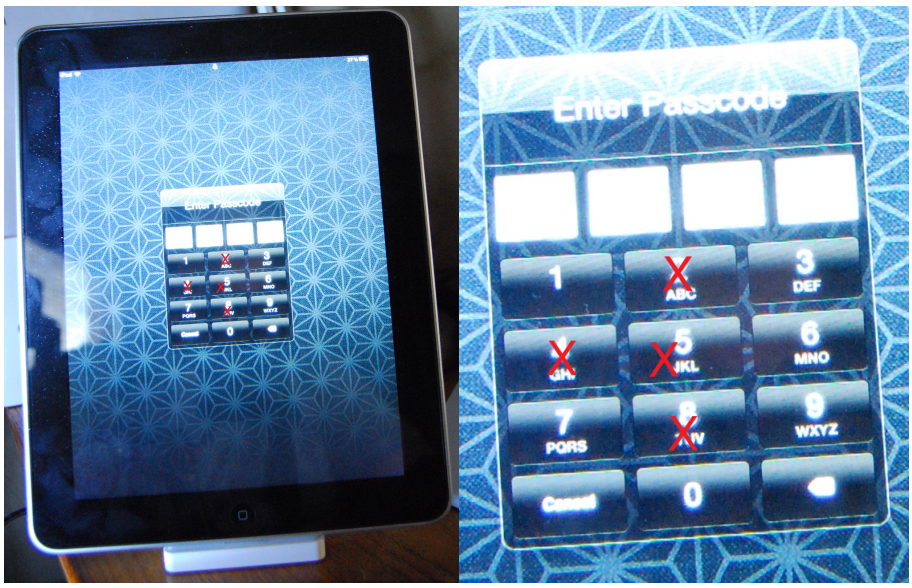
Utilizing Gimp, a second layer was created over top of example 7, with a red X placed in each noted finger print from example 9, to create the following image in example 10 (zoomed to highlight relevant portions):



Example 10

The image of the iPad was then removed from view, to allow the red X fingerprint markers to remain. This was then overlaid onto original pin number login screen

from example 8, to clearly show the suspected pin numbers from the device, in example 11:



Example 11

Superimposing the observed fingerprints allow for a significant reduction in possible pin numbers for successful authentication to the device.

As it is known that for iOS pin number authentication the pin must be 4 characters, we can see that we have 4 candidates to choose from and in this particular case, the number of possible authentication codes (AC) be reduced as such:

Input 1: 4 choices

Input 2: 3 choices

Input 3: 2 choices

Input 4: 1 choice

$AC = 4 \times 3 \times 2 \times 1 = 24$

This significant reduction in possible authentication codes makes unauthorized device entry a trivial exercise.

Need for Alternative Authentication Mechanisms

The speed at which mobile device authentication mechanisms break down is alarming. As was shown previously, this was largely due to the flawed input mechanism that users interact with the device through. Also at fault are the poorly designed authentication systems themselves. As mobile devices are quickly becoming the key to our digital lives, both personal and professional, it is clear that a more secure yet easy to use solution needs to be developed.

The three longstanding paradigms of user authentication are (Grensing-Pophel, 2011):

Paradigm	Example	Common Uses
Something you know	Password Pin Number Pattern	Computer Phone ATM
Something you have	RSA Token SecurID Card	Computer Physical Access
Something you are	Fingerprint Iris	Laptop Physical Access

Chart 1

These paradigms have been in place for millennia, (Singh, 1999) and continue to be pervasive in all restricted-access scenarios. In most current implementations, the "something you have" scenario requires extra hardware, such as SecurID card readers, or keychain token devices (RSA, 2012). These systems provide a strong augmentation to overall security, but are not generally accepted alternatives for mobile devices, as the form factor must be as small as possible, and dedicated hardware technologies are chosen when they can serve multiple functions. These systems are bulky and generally single function. "Something you are" implementations generally suffer from a similar issue; a requirement for a single function piece of hardware dedicated to authentication only.

With current mobile devices however, a wide variety of hardware sensing mechanisms are available that serve a variety of purposes. These mechanisms offer

a wide variety of features that mobile users have become dependent on, such as wireless networking, pictures, global positioning systems, and device-to-device networking. Many of these devices, if applied in the right manner, could serve as either stand-alone or secondary authentication mechanisms. Some of these devices will fit into our traditional three-paradigm authentication set, while others may offer the opportunity for expansion of those paradigms.

The following chart looks provides an overview of currently utilized sensing (Lane, Miluzzo, Lu, Peebles, Choudhury, & Campbell, 2010) in mobile devices, as well as their potential for use in authentication mechanisms:

Sensor /	Description	Authentication Paradigm	Example
Touchscreen	Typically a capacitive touchscreen system. Functions by sensing electrical discharge on contact with human body.	Something you know	Password Pin Pattern Question
Camera(s)	Mobile devices generally utilize 1-2 cameras, either rear facing or front and rear facing, in the 1-10 megapixel range.	Something you are	Facial Recognition Expression Recognition

Microphone(s)	Most devices are equipped with several microphones, designed to help pick up sound and eliminate background noise.	Something you are	Voice Authentication
Accelerometer	An accelerometer is used to measure acceleration and is generally employed in mobile devices to sense the device's own movements, typically for interacting with software applications.	Something you are	Gait Authentication
Gyroscope	A gyroscope is used to measure orientation and is generally employed in mobile devices to sense the device's own movements, typically for interacting with software applications.	Something you are	Gait Authentication
Global **Positioning** **System**	Global positioning systems are used to gather information about the device's physical location, including latitude and longitude, and in some cases altitude, speed, or other metrics about location and movement.	*Somewhere* you are	Location Authentication

Ambient Light	Ambient light sensors are used to detect the light levels around the device, and are typically used to control screen brightness for battery consumption prevention.	None	None
Proximity	Proximity sensors are used to detect nearby objects, typically the human face. These sensors are used to turn off the touchscreen when the device is pressed against the face so as to prevent accidental input to the touch screen.	None	None
Compass	The compass system is similar to a traditional compass in that it provides your overall direction in terms of degrees with 0 degrees being North.	None	None
Near Field Communication	Near field communication is a two-way transmission of data using a similar system as RFID. It is typically used for very short range communications, such as scanning a credit card	None	None

	number.		
BlueTooth	BlueTooth is a wireless microwave frequency used for short range transmission of data (typically up to 30 feet), and is designed for low power consumption.	None	None
802.11x / WiFi	802.11x is a wireless microwave frequency used for long range transmission of data (typically up to 300 feet).	None	None

Chart 2

The less well-known authentication mechanisms above, such as expression recognition, voice authentication, and gait authentication mechanisms are largely based on academic research and have had little widespread implementation in consumer devices.

Expression recognition (Essa, 1997) is similar to facial recognition, but rather than simply analyzing a static image, the algorithm utilizes video capture to analyze both the face and the facial expressions. This type of mechanism provides an added layer of security on to traditional facial recognition systems, such as the one recently released on Android 4.0. This type of authentication seems optimal as it requires no user input. This authentication method, utilizes the front facing camera to

authenticate you via the "something you are" paradigm, by comparing facial features between a trusted photo and the current image sensed by the camera (Lu, 2003). While a novel feature, it took only a short amount of time for a proof of concept exploit to appear, showing that the technology could be fooled by a simple photo of the person shown on the screen of another mobile device (Murphy, 2011). While improvements can be made to the underlying authentication algorithm, such as utilizing expression recognition and blink detection (Pan, Qu, & Sun, 2010), it remains to be seen whether this technology can be made secure.

Voice based authentication systems also show great promise. Voice based authentication can take 2 forms (Why Voice Authentication is the Better Biometric):

- Text Dependent
- Text Independent

In both forms, an audio recording or "fingerprint" of the users voice is taken and stored in a secure fashion on the device. Upon authentication, the user's voice is analyzed against this recording, and an analysis is done to determine if it is the same person. In the text dependent form, the analysis is done on a predetermined and repeated phrase. This phrase is used each time the user authenticates to the device. In the text independent form, the device dynamically listens to the user's voice and picks reference words. Upon authentication, the device may present a number of words to the user for them to repeat and authenticate. Both systems are potentially

vulnerable to the "mimic" attack, that of someone mimicking another person, most likely through the playback of a recording. The text independent form of voice authentication makes the mimic attack slightly more difficult, as the challenge words will change throughout time.

Gait authentication is the analysis of a user's gait to determine who they are (Gafurov, Helkala, & Søndrol, 2006). Gait is defined as "a manner of walking or moving on foot". This type of authentication is done through the use of the gyroscope and accelerometer of a device. The premise is that each human has a unique walking speed and pattern that can be analyzed as an authentication signal. While this type of solution is novel, the process of authentication requires the user to be in motion on foot – not within the typical use case of the great majority of users, typically seated at a desk during the day, or using their mobile device in a car.

Improved Authentication Mechanisms

Based on the testing and research performed above, it is clear that there is a need for improved authentication mechanisms. It is a challenging topic, as it not only has to take into account overall security, but it must be usable enough that it does not alienate the user or distract them too much from their daily lives.

Three solutions are thus proposed below; the first is a proposal for a new authentication system based on newly available sensing mechanisms, and the latter 2 are improvements to the most popular existing authentication mechanisms.

Improvement 1: Somewhere You Are – A New Authentication Paradigm

In the analysis done in chart 2, one particular scheme is proposed that has seen little previous discussion. This is the use of a mobile device's global positioning system (or combination of WiFi and Cellular signals) to authenticate a user based on where they are. This type of solution would allow a user to remain authenticated when they are in "safe zones", such as their home or their office. This would effectively allow a user to bypass all authentication mechanisms when located within a "safe zone", or an area where the user has explicitly whitelisted the area, as presumably mitigating controls are in place, such as physical security.

To extend this functionality, the Bluetooth communication protocol can be used to add "safe device zones". An example of a "safe device zone" is a user's car. The mobile device and car create a pairing based on the previously approved Bluetooth protocol. The Bluetooth transceiver within the car has a unique MAC address that can be used as an authentication key, so the device would enforce no authentication challenge to the user while that user is paired with their car's Bluetooth system.

These solutions would allow a user to use the device unencumbered by authentication requests when in trusted scenarios.

The obvious question is, how does bypassing authentication challenges improve security?

Real-world effectiveness of security is a constant balance between usability and cryptographic strength. If designation of "safe zones" allows a user to significantly reduce their daily authentication efforts, then that same user will be far more likely to tolerate a stronger (and less convenient) authentication system when they are not within a "safe zone", as these events will likely be in the minority of authentication events throughout the day.

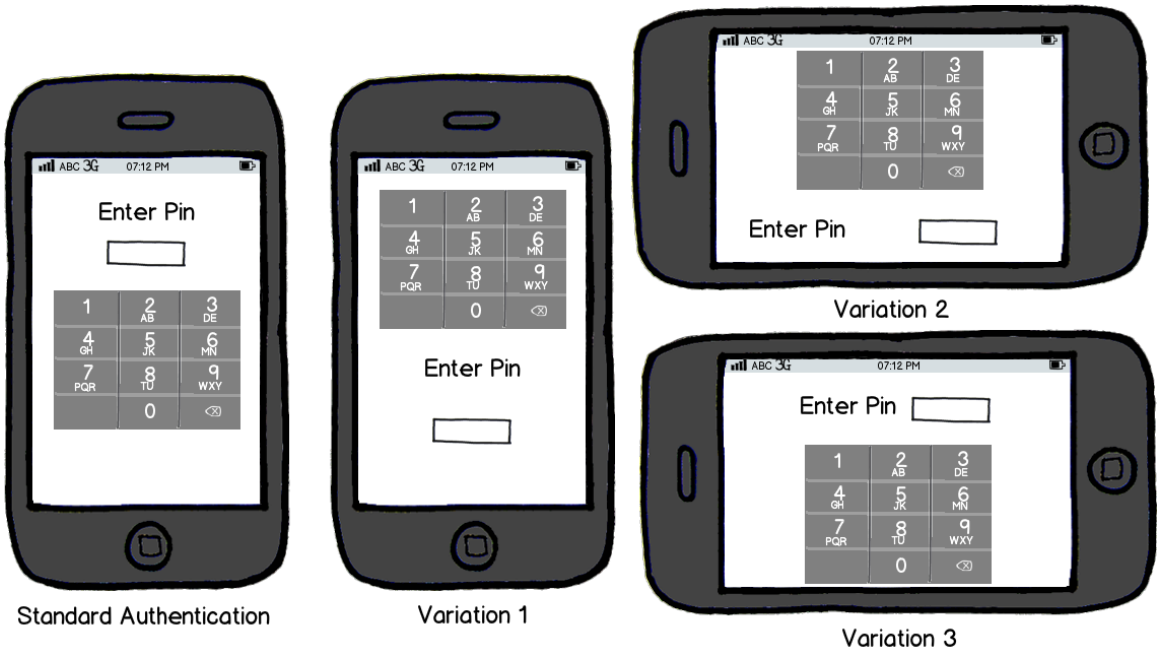
While this example suggests placement within a "safe zone" would allow a total bypass of authentication protocol, it is just as feasible to force a very lightweight authentication system in these scenarios (such as the Android pattern match), with a more robust system when located outside as a "safe zone".

This type of authentication system, if implemented, would not fit into the traditional three authentication paradigms of something you have, something you know, and something you are. For this reason, a new paradigm is proposed; somewhere you are. This paradigm would utilize location based data, typically sensed through a GPS mechanism (or could alternatively be done through 802.11x / WiFi or cellular triangulation in certain areas), to identify "safe zones" where authentication is not required, or could be of a less robust nature.

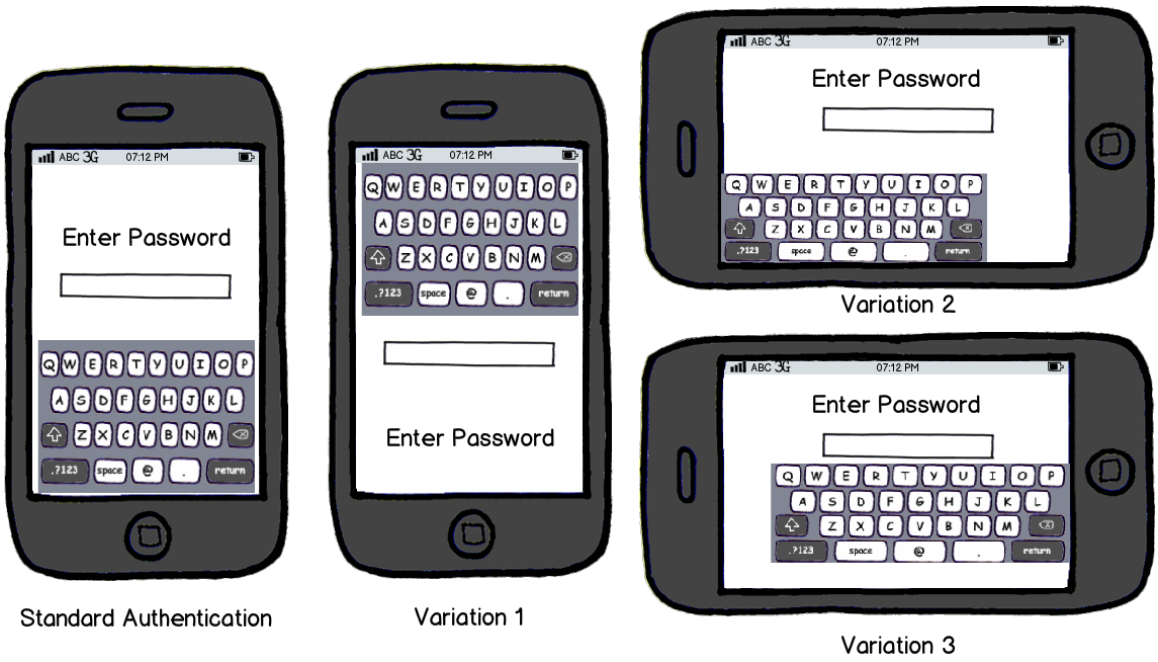
Improvement 2: Forced Rotation and Movement

The authentication mechanisms outlined in examples 1-5 above show systems that are not particularly strong, but in theory provide enough of a challenge to deter the casual infiltrator. However, with the "smudge" attack method outlined later on, the overall security of these systems breaks down very quickly. One simple to implement method to thwart the threat of the "smudge" attack is through forced rotation and movement of the login screen.

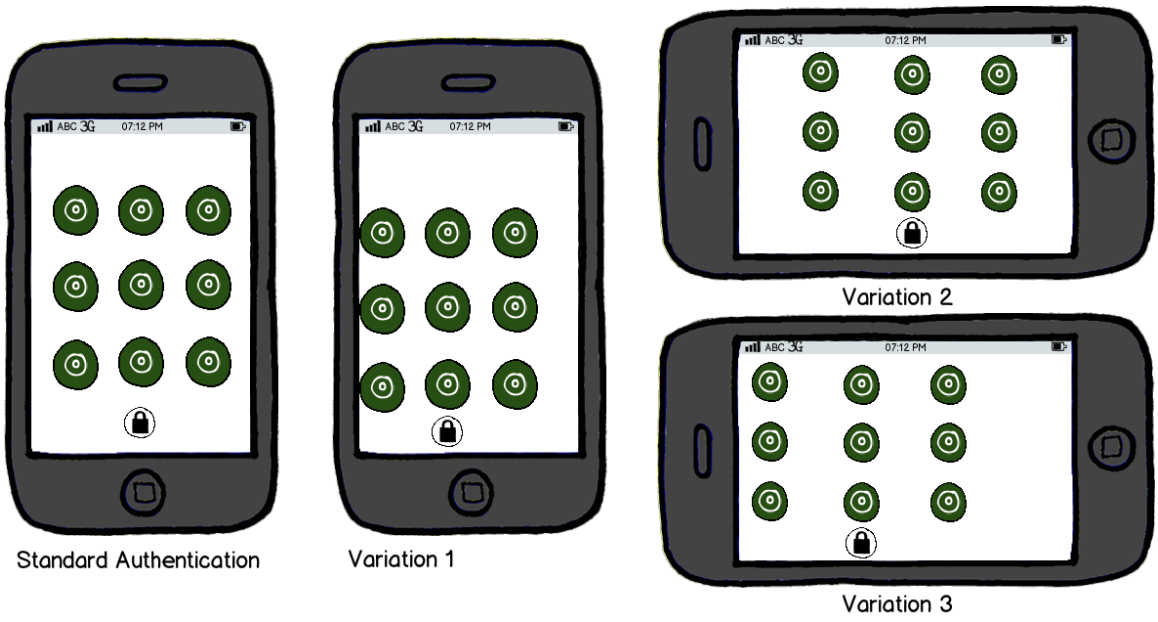
The following mockups show an implementation of the pin number, password, and pattern authentication systems, along with 3 variations, that would be forced and randomized by the device operating system.



Example 12



Example 13



Example 14

Note that in example 14, a reference point is required to ensure the user inputs the pattern from the proper orientation. In the example, the padlock icon was used as that reference point.

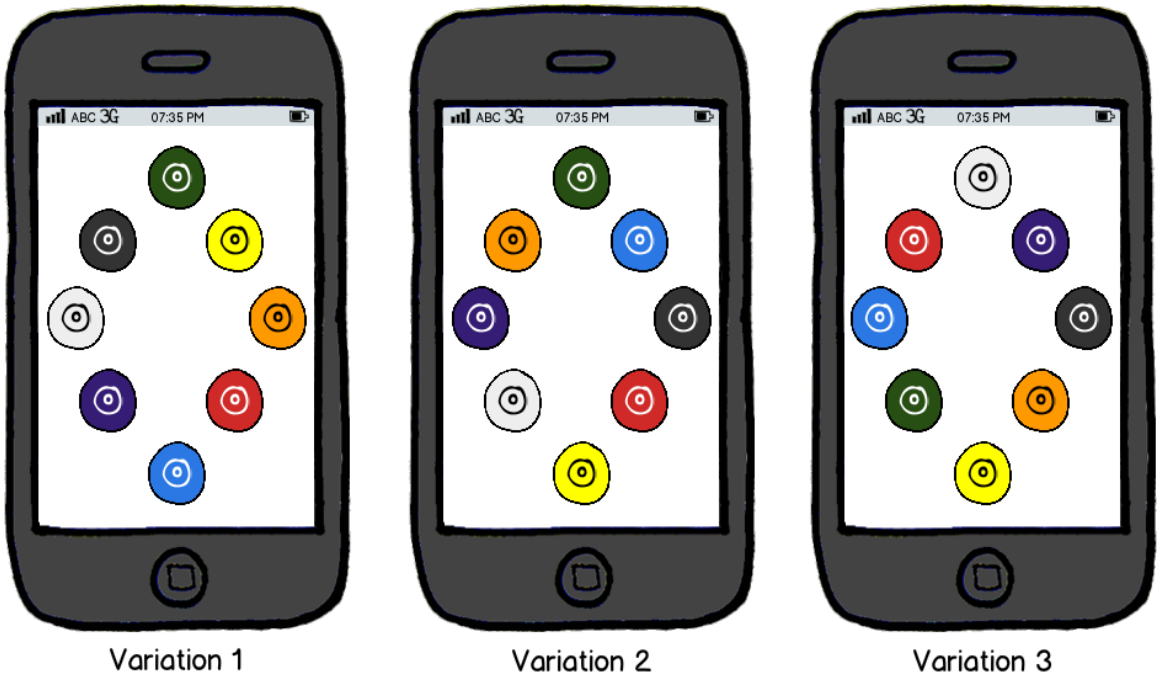
The use of forced rotation and movement of input points would allow for significant obfuscation of smudge points, reducing the ability for a smudge attack to be effective. The forced movement and rotation would create overlapping and intersecting smudges, as well as less smudge buildup in distinct areas of the touchscreen.

Weaknesses

Some weaknesses with this method do exist. The overall effectiveness of the rotation and movement mechanism is increased as the number of authentication attempts since a full cleaning increases. This relies on obfuscation of the true unlock pattern through buildup of finger oil on the screen. While this may be effective, the most recent unlock pattern will generally be most visible as the most recent unlock layer. If the pattern shape can be retrieved, the number of guesses needed to apply that pattern on the right set of nodes (tested horizontally and vertically) may be small enough to make this authentication system similarly vulnerable to the systems previously reviewed.

Improvement 3: Dynamic Patterns

The core issue with pattern-based systems is the excessive residue left behind on the touchscreen. A solution to this problem is to ensure a different pattern is entered on each screen. An example implementation of this is shown below.



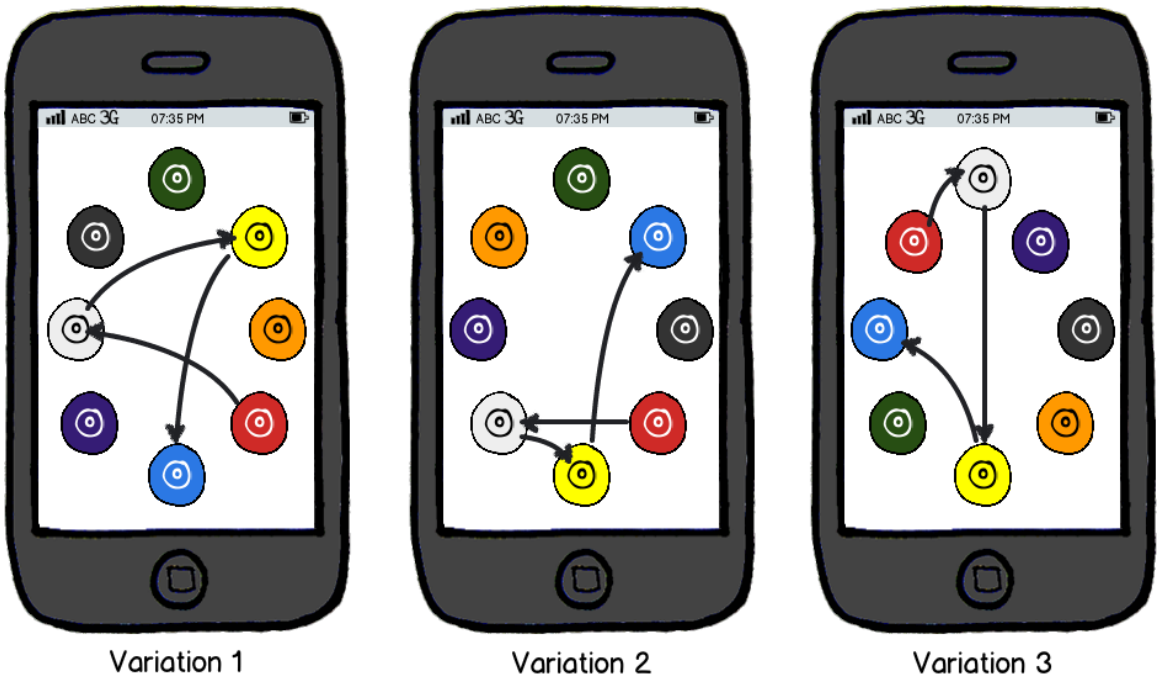
Variation 1 Variation 2 Variation 3

Example 15

This pattern matching system requires assigns a unique color to each node, and randomizes the order of the nodes on each login attempt. This mechanism requires the removal of the center node, as a user must be able to access each node from any point on the screen. This has no significant effect on overall complexity, as the inclusion of a center node (in the current pattern system) that blocks possible

pattern variations also reduces overall complexity. By arranging the available nodes in a circle with no central blocking node, we can also allow reuse of nodes, which will significantly increase the overall keyspace.

In order to authenticate, the user would remember their color pattern. For example, to authenticate to the variations in example 15 with the passcode of Red, White, Yellow, Blue, the following patterns would be entered.



Variation 1 Variation 2 Variation 3

Example 16

It can be seen from this example that the overall smudge lines would be useless for an attacker trying to force authentication. This system would actually increase the overall number of available authentication patterns. Recall the standard Android

pattern authentication system has 389,112 possible patterns, with half of those being technically possible but unlikely and highly prone to entry error. This brings the overall use number of authentication patterns down to 158,410.

To summarize the rules for this system:

- Patterns must be a minimum of 4 nodes and a maximum of 8 nodes
- Nodes may be visited any number of times; however no node may be visited twice consecutively

To calculate the overall complexity of this pattern, it is simply a matter of factorializing the number of distinct nodes. For the example above, with 8 nodes, the number of authentication patterns is:

AC = Number of Nodes * (Number of nodes -1)$^{\text{pattern length -1}}$

AC for 4 Digits = 8*7*7*7 = 8 * 7^3 = 2,744

AC for 5 Digits = 8*7*7*7*7 = 8 * 7^4 = 19,208

AC for 6 Digits = 8*7*7*7*7*7 = 8 * 7^5 = 134,456

AC for 7 Digits = 8*7*7*7*7*7*7 = 8 * 7^6 = 941,192

AC for 8 Digits = 8*7*7*7*7*7*7*7 = 8 * 7^7 = 6,588,344

Sum of authentication patterns for this system: **7,685,944**

Increasing the total number of nodes to 9 creates an even stronger system:

AC for 4 Digits = 9*8*8*8 = $9 * 8^3$ = 4,608

AC for 5 Digits = 9*8*8*8*8 = $9 * 8^4$ = 36,864

AC for 6 Digits = 9*8*8*8*8*8 = $9 * 8^5$ = 294,912

AC for 7 Digits = 9*8*8*8*8*8*8 = $9 * 8^6$ = 2,359,296

AC for 8 Digits = 9*8*8*8*8*8*8*8 = $9 * 8^7$ = 18,874,368

AC for 9 Digits = 9*8*8*8*8*8*8*8*8 = $9 * 8^8$ = 150,994,944

Sum of authentication patterns for this system: **172,564,992**

As has been demonstrated, this alternative system retains a similar overall ease of use swipe mechanism, but increases the available key space by over 4000% from that of the existing pattern matching mechanism.

Conclusion

The mobile device landscape is expanding at a great pace. Smart phones are quickly approaching a 50% stake in the phone market, while tablet computers have finally emerged as a distinct and robust market. With the advent of e-readers, an "in-between" device market has evolved. These devices are aggressively pursuing becoming the do-it-all device that not only replaces the desktop computer, but also serves as the main interface to everyday life tasks such as banking, driving, communication, shopping, and more.

Authentication methods on these devices almost universally rely on touchscreen input to a variety of schemes, ranging from 4 digit pin numbers to authentication patterns, to passwords. These systems provide a minimal level of security for the user. While there is a constant tradeoff between ease of authentication and cryptographic complexity, the pervasive use of a touchscreen as the main input method causes a number of issues. Most significantly, the touchscreen mechanism is prone to forming a buildup of smudges left by the user's fingers. These smudges appear most prominently in areas of the screen where there is the most activity; the authentication pattern is naturally one of the most active areas on the touchscreen, so it is very common for a significant smudge build up to occur in this area, giving the attacker a very easy to reproduce path to authentication. While this is most easily exploitable using the Android pattern matching system, it was shown that the attack is reproducible for a basic pin number system, allowing a significant reduction in the number of possible authentication pins.

It is clear that an improvement is needed in the area of mobile device authentication schemes. Traditionally, the three factors of authentication have been something you know, something you have, and something you are. With the wide array of sensors available in mobile devices, new and potentially usable information is being collected about users and their surroundings that could be utilized for authentication. To demonstrate this, a fourth authentication factor was proposed, *somewhere you are*, that utilizes the GPS sensor information to identify safe-zones

for authentication bypass, with the supposition that decreased security in certain situations would make the user more accepting of stronger authentication mechanisms in cases where location cannot be considered secure.

In addition to using new sensor mechanisms for authentication, improvements can be made to existing systems to reduce or eliminate the chance of a smudge attack. By dynamically placing the authentication puzzle on the touchscreen, through X-Y axis manipulation as well as rotation, any smudges left will be in different spots each time. The downfall of this system is that distances and proportions between taps or swipes can still be retained, giving an attack a bit of useful information about the authentication key. As an additional improvement to this system, an alternative pattern puzzle was proposed that replaces the standard node pattern with a node coloring scheme. This node coloring scheme allows for random placement of colored nodes on screen, with a standard color pattern used as an authentication key. This system ensures a different smudge pattern left on screen each time, while at the same time significantly increasing the overall available key space over the standard Android pattern system.

Bibliography

Andersson, R. (2011, March 28). *Android Lock Screen*. Retrieved from GitHub: https://github.com/rickard2/Android-Lock-Screen

Android Open Source Project. (n.d.). Retrieved February 1, 2012, from Android Open Source Project: http://source.android.com

Aviv, A., Gibson, K., Mossop, E., Blaze, M., & Smith, J. (2010). Smudge Attacks on Smartphone Touch Screens. *WOOT'10 Proceedings of the 4th USENIX conference on Offensive technologies* .

comScore. (2011, October 5). *comScore Reports August 2011 U.S. Mobile Subscriber Market Share*. Retrieved January 2, 2012, from ComScore: www.comscore.com

Currie, D. (2012). *Shedding some light on Voice Authentication.* SANS Institute, Infosec Reading Room. SANS Institute InfoSec Reading Room.

Essa, I. (1997). Coding, Analysis, Interpretation, and Recognition of Facial Expressions. *IEEE Transactions on Pattern Analysis and Machine Intelligence , 19* (7).

Gafurov, D., Helkala, K., & Søndrol, T. (2006). Biometric Gait Authentication Using Accelerometer Sensor. *Journal of Computers , 1* (7), 51-58.

Gallo, F. (2011). *NFC Tags A technical introduction, applications and products.* 1-21: NXP Semiconductors.

Grensing-Pophel, L. (2011). Security and Identity. *EContent , 34*.

Hoover, J. N. (2012, January 17). *NSA Releases Secure Android Version*. Retrieved from Information Week: www.informationweek.com

Kaseorg, A. (2011, February 16). *How many combinations does Android 9 point unlock have?* Retrieved January 27, 2012, from Quora: www.quora.com

Kumparak, G. (2012, February 3). *U.S. Government & Military To Get Secret-Worthy Android Phones*. Retrieved February 4, 2012, from TechCrunch: www.techcrunch.com

Lane, N., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., & Campbell, A. (2010, September). A Survey of Mobile Phone Sensing. *IEEE Communications Magazine* , pp. 140-150.

Lu, X. (2003). *Image Analysis for Face Recognition*. Retrieved from Face-Rec: http://www.face-rec.org/interesting-papers/General/ImAna4FacRcg_lu.pdf

Metro, A. D. *Touch Screen Technology Primer.* A D Metro.

Murphy, D. (2011). *Blogger Breaks Android Face Recognition with... A Picture?* Retrieved from PC Mag: http://www.pcmag.com/article2/0,2817,2396321,00.asp

Nielsen Communications. (2011, September 1). *Smartphone penetration hits 40% of overall U.S. mobile Gift Guide » market*. Retrieved January 27, 2012, from Into Mobile: www.intomobile.com

Nielsen. (2011, November 3). *Generation App: 62% of Mobile Users 25-34 own Smartphones*. Retrieved February 4, 2012, from Nielsen.com: www.nielsen.com

Pan, Qu, & Sun. (2010). Liveness Detection for Face Recognition. *InTech* .

Perez, S. (2011, 9 22). *Gartner: Apple iPad Will Be Top Tablet Through 2014*. Retrieved 1 27, 2012, from TechCrunch: www.techcrunch.com

RSA. (2012). *Securing Your Future with Two-Factor Authentication*. Retrieved from RSA: http://www.rsa.com/node.aspx?id=1156

(1995). *Secure Hash Standard.* Federal Information Processing Standards Publications . FIPS.

Singh, S. (1999). *The Code Book.* New York, NY: Anchor Books.

Smalley, S. (2011, August). The Case for SE Android. National Security Agency.

Usman. (2011, September 17). *A Quick Overview Of Windows 8 Lock Screen*. Retrieved February 5, 2012, from Addictive Tips: www.addictivetips.com

Why Voice Authentication is the Better Biometric. (n.d.). Retrieved January 15, 2012, from Sub K: www.subk.co.in

Appendices

Appendix 1 – Node Object Definition

```
 1 <?php
 2 class Node {
 3
 4 function Node($number){  //CONSTRUCTOR
 5
 6 // Set this nodes number
 7 $this->position = $number;
 8
 9 } // end constructer
10
11 function isavailable($currentpattern)
12 {
13     $this->currentpos = end($currentpattern);
14
15     if (in_array($this->position, $currentpattern))
16     {
17         // If current node already exists in pattern, return false
18         return false;
19     }
20
21     // Center Node
22     if ($this->position == 4)
23     {
24         // Rules for center node
25         // available from all spots anytime, assuming it hasn't been used
26         // which is checked previously
27         return true;
28     }
29
30     // Side Nodes - Vertical
31     if (($this->currentpos == 1 && $this->position == 7) || $this->currentpos == 7 &&
$this->position == 1)
32     {
33         // If moving from node 1 to 7, node 4 must have been visited, else false
34         if (in_array(4, $currentpattern))
35         {
36                 return true;
37         }
38         else {
39              return false;
40         }
41     }
42
43     // Side Nodes - Horizontal
44     if (($this->currentpos == 3 && $this->position == 5) || $this->currentpos == 5 &&
$this->position == 3)
45     {
```

```
46          // If moving from node 1 to 7, node 4 must have been visited, else false
47          if (in_array(4, $currentpattern))
48          {
49               return true;
50          }
51          else {
52               return false;
53          }
54      }
55
56      // Corner Nodes - if coming from other corner node
57      if (($this->position == 0 || $this->position == 2 || $this->position == 6 ||
$this->position == 8) &&
58                ($this->currentpos == 0 || $this->currentpos == 2 || $this->currentpos == 6 ||
$this->currentpos == 8))
59      {
60          // Rules for corner nodes
61          // If traversing from other corner node
62          // currentpos node + this node divided by 2 must be traversed
63          // Example - going from 0->2 requires 1 already visited (0+2 = 2, 2/2 = 1)_
64          // Example - going from 6->2 required 4 already visited (6+2 = 8, 8/2 = 4)
65
66          $midpoint = ($this->position + $this->currentpos) / 2;
67          if(in_array($midpoint, $currentpattern))
68                  {
69               return true;
70          }
71          else {
72              return false;
73          }
74      }
75      // if we made it this far, we've passed all blacklist conditions
76      return true;
77  } // end isavailable method
78
79  } // end class
80  ?>
```

Appendix 2 – Android Pattern Count – Brute Force (Haskell) (Kaseorg, 2011)

```
dots = [(row, col) | row <- [0..2], col <- [0..2]]
line (r, c) (r', c') = takeWhile (/= (r', c')) $
  zip [r, r + div (r' - r) g ..] [c, c + div (c' - c) g ..]
    where g = gcd (r' - r) (c' - c)
extensions pattern@(dot : _) =
  [new : pattern | new <- dots,
   not $ new `elem` pattern, all (`elem` pattern) $ line dot new]
search pattern found = foldr search (pattern : found) $ extensions pattern
valid pattern = length pattern >= 4
main = print . length . filter valid . foldr search [] $ map (: []) dots

// Output: 389112
```

Appendix 3 – LockPatternUtils.java

```
/*
 * Copyright (C) 2007 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.android.internal.widget;

import android.app.admin.DevicePolicyManager;
import android.content.ContentResolver;
import android.content.Context;
import android.database.Cursor;
import android.net.Uri;
import android.os.FileObserver;
import android.os.RemoteException;
import android.os.ServiceManager;
import android.os.SystemClock;
import android.provider.Calendar;
import android.provider.Settings;
import android.security.MessageDigest;
import android.telephony.TelephonyManager;
import android.text.TextUtils;
import android.text.format.DateFormat;
import android.util.Log;
import android.widget.Button;

import com.android.internal.R;
import com.android.internal.telephony.ITelephony;

import com.google.android.collect.Lists;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.util.List;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.TimeZone;

/**
 * Utilities for the lock patten and its settings.
 */
public class LockPatternUtils {

    private static final String TAG = "LockPatternUtils";

    private static final String SYSTEM_DIRECTORY = "/system/";
    private static final String LOCK_PATTERN_FILE = "gesture.key";
    private static final String LOCK_PASSWORD_FILE = "password.key";
```

```
    /**
     * The maximum number of incorrect attempts before the user is prevented
     * from trying again for {@link #FAILED_ATTEMPT_TIMEOUT_MS}.
     */
    public static final int FAILED_ATTEMPTS_BEFORE_TIMEOUT = 5;

    /**
     * The number of incorrect attempts before which we fall back on an alternative
     * method of verifying the user, and resetting their lock pattern.
     */
    public static final int FAILED_ATTEMPTS_BEFORE_RESET = 20;

    /**
     * How long the user is prevented from trying again after entering the
     * wrong pattern too many times.
     */
    public static final long FAILED_ATTEMPT_TIMEOUT_MS = 30000L;

    /**
     * The interval of the countdown for showing progress of the lockout.
     */
    public static final long FAILED_ATTEMPT_COUNTDOWN_INTERVAL_MS = 1000L;

    /**
     * The minimum number of dots in a valid pattern.
     */
    public static final int MIN_LOCK_PATTERN_SIZE = 4;

    /**
     * The minimum number of dots the user must include in a wrong pattern
     * attempt for it to be counted against the counts that affect
     * {@link #FAILED_ATTEMPTS_BEFORE_TIMEOUT} and {@link #FAILED_ATTEMPTS_BEFORE_RESET}
     */
    public static final int MIN_PATTERN_REGISTER_FAIL = 3;

    private final static String LOCKOUT_PERMANENT_KEY =
"lockscreen.lockedoutpermanently";
    private final static String LOCKOUT_ATTEMPT_DEADLINE =
"lockscreen.lockoutattemptdeadline";
    private final static String PATTERN_EVER_CHOSEN_KEY = "lockscreen.patterneverchosen";
    public final static String PASSWORD_TYPE_KEY = "lockscreen.password_type";
    private final static String LOCK_PASSWORD_SALT_KEY = "lockscreen.password_salt";

    private final Context mContext;
    private final ContentResolver mContentResolver;
    private DevicePolicyManager mDevicePolicyManager;
    private static String sLockPatternFilename;
    private static String sLockPasswordFilename;

    private static final AtomicBoolean sHaveNonZeroPatternFile = new
AtomicBoolean(false);
    private static final AtomicBoolean sHaveNonZeroPasswordFile = new
AtomicBoolean(false);
    private static FileObserver sPasswordObserver;

    public DevicePolicyManager getDevicePolicyManager() {
        if (mDevicePolicyManager == null) {
            mDevicePolicyManager =

(DevicePolicyManager)mContext.getSystemService(Context.DEVICE_POLICY_SERVICE);
            if (mDevicePolicyManager == null) {
                Log.e(TAG, "Can't get DevicePolicyManagerService: is it running?",
                        new IllegalStateException("Stack trace:"));
            }
        }
        return mDevicePolicyManager;
    }
    /**
     * @param contentResolver Used to look up and save settings.
```

```
     */
    public LockPatternUtils(Context context) {
        mContext = context;
        mContentResolver = context.getContentResolver();

        // Initialize the location of gesture & PIN lock files
        if (sLockPatternFilename == null) {
            String dataSystemDirectory =
                    android.os.Environment.getDataDirectory().getAbsolutePath() +
                    SYSTEM_DIRECTORY;
            sLockPatternFilename =  dataSystemDirectory + LOCK_PATTERN_FILE;
            sLockPasswordFilename = dataSystemDirectory + LOCK_PASSWORD_FILE;
            sHaveNonZeroPatternFile.set(new File(sLockPatternFilename).length() > 0);
            sHaveNonZeroPasswordFile.set(new File(sLockPasswordFilename).length() > 0);
            int fileObserverMask = FileObserver.CLOSE_WRITE | FileObserver.DELETE |
                    FileObserver.MOVED_TO | FileObserver.CREATE;
            sPasswordObserver = new FileObserver(dataSystemDirectory, fileObserverMask) {
                    public void onEvent(int event, String path) {
                        if (LOCK_PATTERN_FILE.equals(path)) {
                            Log.d(TAG, "lock pattern file changed");
                            sHaveNonZeroPatternFile.set(new
File(sLockPatternFilename).length() > 0);
                        } else if (LOCK_PASSWORD_FILE.equals(path)) {
                            Log.d(TAG, "lock password file changed");
                            sHaveNonZeroPasswordFile.set(new
File(sLockPasswordFilename).length() > 0);
                        }
                    }
                };
            sPasswordObserver.startWatching();
        }
    }

    public int getRequestedMinimumPasswordLength() {
        return getDevicePolicyManager().getPasswordMinimumLength(null);
    }


    /**
     * Gets the device policy password mode. If the mode is non-specific, returns
     * MODE_PATTERN which allows the user to choose anything.
     */
    public int getRequestedPasswordQuality() {
        return getDevicePolicyManager().getPasswordQuality(null);
    }

    /**
     * Returns the actual password mode, as set by keyguard after updating the password.
     *
     * @return
     */
    public void reportFailedPasswordAttempt() {
        getDevicePolicyManager().reportFailedPasswordAttempt();
    }

    public void reportSuccessfulPasswordAttempt() {
        getDevicePolicyManager().reportSuccessfulPasswordAttempt();
    }

    /**
     * Check to see if a pattern matches the saved pattern.  If no pattern exists,
     * always returns true.
     * @param pattern The pattern to check.
     * @return Whether the pattern matches the stored one.
     */
    public boolean checkPattern(List<LockPatternView.Cell> pattern) {
        try {
            // Read all the bytes from the file
            RandomAccessFile raf = new RandomAccessFile(sLockPatternFilename, "r");
```

```
            final byte[] stored = new byte[(int) raf.length()];
            int got = raf.read(stored, 0, stored.length);
            raf.close();
            if (got <= 0) {
                return true;
            }
            // Compare the hash from the file with the entered pattern's hash
            return Arrays.equals(stored, LockPatternUtils.patternToHash(pattern));
        } catch (FileNotFoundException fnfe) {
            return true;
        } catch (IOException ioe) {
            return true;
        }
    }

    /**
     * Check to see if a password matches the saved password.  If no password exists,
     * always returns true.
     * @param password The password to check.
     * @return Whether the password matches the stored one.
     */
    public boolean checkPassword(String password) {
        try {
            // Read all the bytes from the file
            RandomAccessFile raf = new RandomAccessFile(sLockPasswordFilename, "r");
            final byte[] stored = new byte[(int) raf.length()];
            int got = raf.read(stored, 0, stored.length);
            raf.close();
            if (got <= 0) {
                return true;
            }
            // Compare the hash from the file with the entered password's hash
            return Arrays.equals(stored, passwordToHash(password));
        } catch (FileNotFoundException fnfe) {
            return true;
        } catch (IOException ioe) {
            return true;
        }
    }

    /**
     * Check to see if the user has stored a lock pattern.
     * @return Whether a saved pattern exists.
     */
    public boolean savedPatternExists() {
        return sHaveNonZeroPatternFile.get();
    }

    /**
     * Check to see if the user has stored a lock pattern.
     * @return Whether a saved pattern exists.
     */
    public boolean savedPasswordExists() {
        return sHaveNonZeroPasswordFile.get();
    }

    /**
     * Return true if the user has ever chosen a pattern.  This is true even if the
pattern is
     * currently cleared.
     *
     * @return True if the user has ever chosen a pattern.
     */
    public boolean isPatternEverChosen() {
        return getBoolean(PATTERN_EVER_CHOSEN_KEY);
    }

    /**
     * Used by device policy manager to validate the current password
```

```
     * information it has.
     */
    public int getActivePasswordQuality() {
        int activePasswordQuality = DevicePolicyManager.PASSWORD_QUALITY_UNSPECIFIED;
        switch (getKeyguardStoredPasswordQuality()) {
            case DevicePolicyManager.PASSWORD_QUALITY_SOMETHING:
                if (isLockPatternEnabled()) {
                    activePasswordQuality =
DevicePolicyManager.PASSWORD_QUALITY_SOMETHING;
                }
                break;
            case DevicePolicyManager.PASSWORD_QUALITY_NUMERIC:
                if (isLockPasswordEnabled()) {
                    activePasswordQuality = DevicePolicyManager.PASSWORD_QUALITY_NUMERIC;
                }
                break;
            case DevicePolicyManager.PASSWORD_QUALITY_ALPHABETIC:
                if (isLockPasswordEnabled()) {
                    activePasswordQuality =
DevicePolicyManager.PASSWORD_QUALITY_ALPHABETIC;
                }
                break;
            case DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC:
                if (isLockPasswordEnabled()) {
                    activePasswordQuality =
DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC;
                }
                break;
        }
        return activePasswordQuality;
    }

    /**
     * Clear any lock pattern or password.
     */
    public void clearLock() {
        getDevicePolicyManager().setActivePasswordState(
                DevicePolicyManager.PASSWORD_QUALITY_UNSPECIFIED, 0);
        saveLockPassword(null, DevicePolicyManager.PASSWORD_QUALITY_SOMETHING);
        setLockPatternEnabled(false);
        saveLockPattern(null);
        setLong(PASSWORD_TYPE_KEY, DevicePolicyManager.PASSWORD_QUALITY_SOMETHING);
    }

    /**
     * Save a lock pattern.
     * @param pattern The new pattern to save.
     */
    public void saveLockPattern(List<LockPatternView.Cell> pattern) {
        // Compute the hash
        final byte[] hash  = LockPatternUtils.patternToHash(pattern);
        try {
            // Write the hash to file
            RandomAccessFile raf = new RandomAccessFile(sLockPatternFilename, "rw");
            // Truncate the file if pattern is null, to clear the lock
            if (pattern == null) {
                raf.setLength(0);
            } else {
                raf.write(hash, 0, hash.length);
            }
            raf.close();
            DevicePolicyManager dpm = getDevicePolicyManager();
            if (pattern != null) {
                setBoolean(PATTERN_EVER_CHOSEN_KEY, true);
                setLong(PASSWORD_TYPE_KEY,
DevicePolicyManager.PASSWORD_QUALITY_SOMETHING);
                dpm.setActivePasswordState(
                        DevicePolicyManager.PASSWORD_QUALITY_SOMETHING, pattern.size());
            } else {
```

```
                dpm.setActivePasswordState(
                        DevicePolicyManager.PASSWORD_QUALITY_UNSPECIFIED, 0);
            }
        } catch (FileNotFoundException fnfe) {
            // Cant do much, unless we want to fail over to using the settings provider
            Log.e(TAG, "Unable to save lock pattern to " + sLockPatternFilename);
        } catch (IOException ioe) {
            // Cant do much
            Log.e(TAG, "Unable to save lock pattern to " + sLockPatternFilename);
        }
    }

    /**
     * Compute the password quality from the given password string.
     */
    static public int computePasswordQuality(String password) {
        boolean hasDigit = false;
        boolean hasNonDigit = false;
        final int len = password.length();
        for (int i = 0; i < len; i++) {
            if (Character.isDigit(password.charAt(i))) {
                hasDigit = true;
            } else {
                hasNonDigit = true;
            }
        }

        if (hasNonDigit && hasDigit) {
            return DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC;
        }
        if (hasNonDigit) {
            return DevicePolicyManager.PASSWORD_QUALITY_ALPHABETIC;
        }
        if (hasDigit) {
            return DevicePolicyManager.PASSWORD_QUALITY_NUMERIC;
        }
        return DevicePolicyManager.PASSWORD_QUALITY_UNSPECIFIED;
    }

    /**
     * Save a lock password.  Does not ensure that the password is as good
     * as the requested mode, but will adjust the mode to be as good as the
     * pattern.
     * @param password The password to save
     * @param quality {@see
DevicePolicyManager#getPasswordQuality(android.content.ComponentName)}
     */
    public void saveLockPassword(String password, int quality) {
        // Compute the hash
        final byte[] hash = passwordToHash(password);
        try {
            // Write the hash to file
            RandomAccessFile raf = new RandomAccessFile(sLockPasswordFilename, "rw");
            // Truncate the file if pattern is null, to clear the lock
            if (password == null) {
                raf.setLength(0);
            } else {
                raf.write(hash, 0, hash.length);
            }
            raf.close();
            DevicePolicyManager dpm = getDevicePolicyManager();
            if (password != null) {
                int computedQuality = computePasswordQuality(password);
                setLong(PASSWORD_TYPE_KEY, computedQuality);
                if (computedQuality != DevicePolicyManager.PASSWORD_QUALITY_UNSPECIFIED)
{
                    dpm.setActivePasswordState(computedQuality, password.length());
                } else {
                    // The password is not anything.
```

```
                    dpm.setActivePasswordState(
                            DevicePolicyManager.PASSWORD_QUALITY_UNSPECIFIED, 0);
                }
            } else {
                dpm.setActivePasswordState(
                        DevicePolicyManager.PASSWORD_QUALITY_UNSPECIFIED, 0);
            }
        } catch (FileNotFoundException fnfe) {
            // Cant do much, unless we want to fail over to using the settings provider
            Log.e(TAG, "Unable to save lock pattern to " + sLockPasswordFilename);
        } catch (IOException ioe) {
            // Cant do much
            Log.e(TAG, "Unable to save lock pattern to " + sLockPasswordFilename);
        }
    }

    /**
     * Retrieves the quality mode we're in.
     * {@see DevicePolicyManager#getPasswordQuality(android.content.ComponentName)}
     *
     * @return stored password quality
     */
    public int getKeyguardStoredPasswordQuality() {
        return (int) getLong(PASSWORD_TYPE_KEY,
DevicePolicyManager.PASSWORD_QUALITY_SOMETHING);
    }

    /**
     * Deserialize a pattern.
     * @param string The pattern serialized with {@link #patternToString}
     * @return The pattern.
     */
    public static List<LockPatternView.Cell> stringToPattern(String string) {
        List<LockPatternView.Cell> result = Lists.newArrayList();

        final byte[] bytes = string.getBytes();
        for (int i = 0; i < bytes.length; i++) {
            byte b = bytes[i];
            result.add(LockPatternView.Cell.of(b / 3, b % 3));
        }
        return result;
    }

    /**
     * Serialize a pattern.
     * @param pattern The pattern.
     * @return The pattern in string form.
     */
    public static String patternToString(List<LockPatternView.Cell> pattern) {
        if (pattern == null) {
            return "";
        }
        final int patternSize = pattern.size();

        byte[] res = new byte[patternSize];
        for (int i = 0; i < patternSize; i++) {
            LockPatternView.Cell cell = pattern.get(i);
            res[i] = (byte) (cell.getRow() * 3 + cell.getColumn());
        }
        return new String(res);
    }

    /*
     * Generate an SHA-1 hash for the pattern. Not the most secure, but it is
     * at least a second level of protection. First level is that the file
     * is in a location only readable by the system process.
     * @param pattern the gesture pattern.
     * @return the hash of the pattern in a byte array.
     */
```

```
    private static byte[] patternToHash(List<LockPatternView.Cell> pattern) {
        if (pattern == null) {
            return null;
        }

        final int patternSize = pattern.size();
        byte[] res = new byte[patternSize];
        for (int i = 0; i < patternSize; i++) {
            LockPatternView.Cell cell = pattern.get(i);
            res[i] = (byte) (cell.getRow() * 3 + cell.getColumn());
        }
        try {
            MessageDigest md = MessageDigest.getInstance("SHA-1");
            byte[] hash = md.digest(res);
            return hash;
        } catch (NoSuchAlgorithmException nsa) {
            return res;
        }
    }

    private String getSalt() {
        long salt = getLong(LOCK_PASSWORD_SALT_KEY, 0);
        if (salt == 0) {
            try {
                salt = SecureRandom.getInstance("SHA1PRNG").nextLong();
                setLong(LOCK_PASSWORD_SALT_KEY, salt);
                Log.v(TAG, "Initialized lock password salt");
            } catch (NoSuchAlgorithmException e) {
                // Throw an exception rather than storing a password we'll never be able
to recover
                throw new IllegalStateException("Couldn't get SecureRandom number", e);
            }
        }
        return Long.toHexString(salt);
    }

    /*
     * Generate a hash for the given password. To avoid brute force attacks, we use a
salted hash.
     * Not the most secure, but it is at least a second level of protection. First level
is that
     * the file is in a location only readable by the system process.
     * @param password the gesture pattern.
     * @return the hash of the pattern in a byte array.
     */
     public byte[] passwordToHash(String password) {
        if (password == null) {
            return null;
        }
        String algo = null;
        byte[] hashed = null;
        try {
            byte[] saltedPassword = (password + getSalt()).getBytes();
            byte[] sha1 = MessageDigest.getInstance(algo = "SHA-
1").digest(saltedPassword);
            byte[] md5 = MessageDigest.getInstance(algo = "MD5").digest(saltedPassword);
            hashed = toHex(sha1, md5);
        } catch (NoSuchAlgorithmException e) {
            Log.w(TAG, "Failed to encode string because of missing algorithm: " + algo);
        }
        return hashed;
    }

    private static final byte[] HEX_CHARS = new byte[]{
            '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E',
'F'
    };

    private static byte[] toHex(final byte[] array1, final byte[] array2) {
```

```
        final byte[] result = new byte[(array1.length + array2.length) * 2];
        int i = 0;
        for (final byte b : array1) {
            result[i++] = HEX_CHARS[(b >> 4) & 0xf];
            result[i++] = HEX_CHARS[b & 0xf];
        }
        for (final byte b : array2) {
            result[i++] = HEX_CHARS[(b >> 4) & 0xf];
            result[i++] = HEX_CHARS[b & 0xf];
        }
        return result;
    }

    /**
     * @return Whether the lock password is enabled.
     */
    public boolean isLockPasswordEnabled() {
        long mode = getLong(PASSWORD_TYPE_KEY, 0);
        return savedPasswordExists() &&
                (mode == DevicePolicyManager.PASSWORD_QUALITY_ALPHABETIC
                        || mode == DevicePolicyManager.PASSWORD_QUALITY_NUMERIC
                        || mode == DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC);
    }

    /**
     * @return Whether the lock pattern is enabled.
     */
    public boolean isLockPatternEnabled() {
        return getBoolean(Settings.Secure.LOCK_PATTERN_ENABLED)
                && getLong(PASSWORD_TYPE_KEY,
DevicePolicyManager.PASSWORD_QUALITY_SOMETHING)
                        == DevicePolicyManager.PASSWORD_QUALITY_SOMETHING;
    }

    /**
     * Set whether the lock pattern is enabled.
     */
    public void setLockPatternEnabled(boolean enabled) {
        setBoolean(Settings.Secure.LOCK_PATTERN_ENABLED, enabled);
    }

    /**
     * @return Whether the visible pattern is enabled.
     */
    public boolean isVisiblePatternEnabled() {
        return getBoolean(Settings.Secure.LOCK_PATTERN_VISIBLE);
    }

    /**
     * Set whether the visible pattern is enabled.
     */
    public void setVisiblePatternEnabled(boolean enabled) {
        setBoolean(Settings.Secure.LOCK_PATTERN_VISIBLE, enabled);
    }

    /**
     * @return Whether tactile feedback for the pattern is enabled.
     */
    public boolean isTactileFeedbackEnabled() {
        return getBoolean(Settings.Secure.LOCK_PATTERN_TACTILE_FEEDBACK_ENABLED);
    }

    /**
     * Set whether tactile feedback for the pattern is enabled.
     */
    public void setTactileFeedbackEnabled(boolean enabled) {
        setBoolean(Settings.Secure.LOCK_PATTERN_TACTILE_FEEDBACK_ENABLED, enabled);
    }
```

```
public void setVisibleDotsEnabled(boolean enabled) {
    setBoolean(Settings.Secure.LOCK_DOTS_VISIBLE, enabled);
}

public boolean isVisibleDotsEnabled() {
    return getBoolean(Settings.Secure.LOCK_DOTS_VISIBLE, true);
}

public void setShowErrorPath(boolean enabled) {
    setBoolean(Settings.Secure.LOCK_SHOW_ERROR_PATH, enabled);
}

public boolean isShowErrorPath() {
    return getBoolean(Settings.Secure.LOCK_SHOW_ERROR_PATH, true);
}

public void setShowCustomMsg(boolean enabled) {
    setBoolean(Settings.Secure.LOCK_SHOW_CUSTOM_MSG, enabled);
}

public boolean isShowCustomMsg() {
    return getBoolean(Settings.Secure.LOCK_SHOW_CUSTOM_MSG, false);
}

public void setCustomMsg(String msg) {
    setString(Settings.Secure.LOCK_CUSTOM_MSG, msg);
}

public String getCustomMsg() {
    return getString(Settings.Secure.LOCK_CUSTOM_MSG);
}

public int getIncorrectDelay() {
    return getInt(Settings.Secure.LOCK_INCORRECT_DELAY, 2000);
}

public void setIncorrectDelay(int delay) {
    setInt(Settings.Secure.LOCK_INCORRECT_DELAY, delay);
}

public void setShowUnlockMsg(boolean enabled) {
    setBoolean(Settings.Secure.SHOW_UNLOCK_TEXT, enabled);
}

public boolean isShowUnlockMsg() {
    return getBoolean(Settings.Secure.SHOW_UNLOCK_TEXT, true);
}

public void setShowUnlockErrMsg(boolean enabled) {
    setBoolean(Settings.Secure.SHOW_UNLOCK_ERR_TEXT, enabled);
}

public boolean isShowUnlockErrMsg() {
    return getBoolean(Settings.Secure.SHOW_UNLOCK_ERR_TEXT, true);
}

/**
 * Set and store the lockout deadline, meaning the user can't attempt his/her unlock
 * pattern until the deadline has passed.
 * @return the chosen deadline.
 */
public long setLockoutAttemptDeadline() {
    final long deadline = SystemClock.elapsedRealtime() + FAILED_ATTEMPT_TIMEOUT_MS;
    setLong(LOCKOUT_ATTEMPT_DEADLINE, deadline);
    return deadline;
}

/**
 * @return The elapsed time in millis in the future when the user is allowed to
```

```
     *   attempt to enter his/her lock pattern, or 0 if the user is welcome to
     *   enter a pattern.
     */
    public long getLockoutAttemptDeadline() {
        final long deadline = getLong(LOCKOUT_ATTEMPT_DEADLINE, 0L);
        final long now = SystemClock.elapsedRealtime();
        if (deadline < now || deadline > (now + FAILED_ATTEMPT_TIMEOUT_MS)) {
            return 0L;
        }
        return deadline;
    }

    /**
     * @return Whether the user is permanently locked out until they verify their
     *   credentials.  Occurs after {@link #FAILED_ATTEMPTS_BEFORE_RESET} failed
     *   attempts.
     */
    public boolean isPermanentlyLocked() {
        return getBoolean(LOCKOUT_PERMANENT_KEY);
    }

    /**
     * Set the state of whether the device is permanently locked, meaning the user
     * must authenticate via other means.
     *
     * @param locked Whether the user is permanently locked out until they verify their
     *   credentials.  Occurs after {@link #FAILED_ATTEMPTS_BEFORE_RESET} failed
     *   attempts.
     */
    public void setPermanentlyLocked(boolean locked) {
        setBoolean(LOCKOUT_PERMANENT_KEY, locked);
    }

    /**
     * @return A formatted string of the next alarm (for showing on the lock screen),
     *   or null if there is no next alarm.
     */
    public String getNextAlarm() {
        String nextAlarm = Settings.System.getString(mContentResolver,
                Settings.System.NEXT_ALARM_FORMATTED);
        if (nextAlarm == null || TextUtils.isEmpty(nextAlarm)) {
            return null;
        }
        return nextAlarm;
    }

    /**
     * @return A formatted string of the next calendar event with a reminder
     * (for showing on the lock screen), or null if there is no next event
     * within a certain look-ahead time.
     */
    public String getNextCalendarAlarm(long lookahead, String[] calendars,
            boolean remindersOnly) {
        long now = System.currentTimeMillis();
        long later = now + lookahead;

        StringBuilder where = new StringBuilder();
        if (remindersOnly) {
            where.append(Calendar.EventsColumns.HAS_ALARM + "=1");
        }
        if (calendars != null && calendars.length > 0) {
            if (remindersOnly) {
                where.append(" AND ");
            }
            where.append(Calendar.EventsColumns.CALENDAR_ID + " in (");
            for (int i = 0; i < calendars.length; i++) {
                where.append(calendars[i]);
                if (i != calendars.length - 1) {
                    where.append(",");
```

```
                }
            }
            where.append(") ");
        }

        String[] projection = new String[] {
            Calendar.EventsColumns.TITLE,
            Calendar.Instances.BEGIN,
            Calendar.EventsColumns.DESCRIPTION,
            Calendar.EventsColumns.EVENT_LOCATION,
            Calendar.EventsColumns.ALL_DAY
        };

        Uri uri = Uri.withAppendedPath(Calendar.Instances.CONTENT_URI,
                String.format("%d/%d", now, later));
        String nextCalendarAlarm = null;
        Cursor cursor = null;

        try {
            cursor = mContentResolver.query(uri,
                    projection, where.toString(), null,
                    Calendar.Instances.DEFAULT_SORT_ORDER);

            if (cursor != null && cursor.moveToFirst()) {

                String title = cursor.getString(0);
                long begin = cursor.getLong(1);
                String description = cursor.getString(2);
                String location = cursor.getString(3);
                boolean allDay = cursor.getInt(4) != 0;

                // Check the next event in the case of allday event. As UTC is used for
allday
                // events, the next event may be the one that actually starts sooner
                if (allDay && !cursor.isLast()) {
                    cursor.moveToNext();
                    long nextBegin = cursor.getLong(1);
                    if (nextBegin < begin + TimeZone.getDefault().getOffset(begin)) {
                        title = cursor.getString(0);
                        begin = nextBegin;
                        description = cursor.getString(2);
                        location = cursor.getString(3);
                        allDay = cursor.getInt(4) != 0;
                    }
                }

                Date start = new Date(begin);
                StringBuilder sb = new StringBuilder();

                if (allDay) {
                    SimpleDateFormat sdf = new SimpleDateFormat(
                            mContext.getString(R.string.abbrev_wday_month_day_no_year));
                    // Calendar stores all-day events in UTC -- setting the timezone
ensures
                    // the correct date is shown.
                    sdf.setTimeZone(TimeZone.getTimeZone("UTC"));
                    sb.append(sdf.format(start));
                } else {
                    sb.append(DateFormat.format("E", start));
                    sb.append(" ");
                    sb.append(DateFormat.getTimeFormat(mContext).format(start));
                }

                sb.append(" ");
                sb.append(title);

                int showLocation = Settings.System.getInt(mContext.getContentResolver(),
                            Settings.System.LOCKSCREEN_CALENDAR_SHOW_LOCATION, 0);
```

```
                if (showLocation != 0 && !TextUtils.isEmpty(location)) {
                    switch(showLocation) {
                        case 1:
                            // Show first line
                            int end = location.indexOf('\n');
                            if(end == -1) {
                                sb.append("\n" + location);
                            } else {
                                sb.append("\n" + location.substring(0, end));
                            }
                            break;
                        case 2:
                            // Show all
                            sb.append("\n" + location);
                            break;
                    }
                }

                int showDescription =
Settings.System.getInt(mContext.getContentResolver(),
                            Settings.System.LOCKSCREEN_CALENDAR_SHOW_DESCRIPTION, 0);

                if (showDescription != 0 && !TextUtils.isEmpty(description)) {
                    switch(showDescription) {
                        case 1:
                            // Show first line
                            int end = description.indexOf('\n');
                            if(end == -1) {
                                sb.append("\n" + description);
                            } else {
                                sb.append("\n" + description.substring(0, end));
                            }
                            break;
                        case 2:
                            // Show all
                            sb.append("\n" + description);
                            break;
                    }
                }

                nextCalendarAlarm = sb.toString();
            }
        } finally {
            if (cursor != null) {
                cursor.close();
            }
        }
        return nextCalendarAlarm;
    }

    private boolean getBoolean(String secureSettingKey) {
        return 1 ==
                android.provider.Settings.Secure.getInt(mContentResolver,
secureSettingKey, 0);
    }

    private boolean getBoolean(String systemSettingKey, boolean defaultValue) {
        return 1 ==
                android.provider.Settings.Secure.getInt(
                        mContentResolver,
                        systemSettingKey, defaultValue ? 1 : 0);
    }

    private void setBoolean(String secureSettingKey, boolean enabled) {
        android.provider.Settings.Secure.putInt(mContentResolver, secureSettingKey,
                                                enabled ? 1 : 0);
    }

    private long getLong(String secureSettingKey, long def) {
```

```
        return android.provider.Settings.Secure.getLong(mContentResolver,
secureSettingKey, def);
    }

    private void setLong(String secureSettingKey, long value) {
        android.provider.Settings.Secure.putLong(mContentResolver, secureSettingKey,
value);
    }

    private int getInt(String systemSettingKey, int def) {
        return android.provider.Settings.Secure.getInt(mContentResolver,
systemSettingKey, def);
    }

    private void setInt(String systemSettingKey, int value) {
        android.provider.Settings.Secure.putInt(mContentResolver, systemSettingKey,
value);
    }

    private String getString(String systemSettingKey) {
        String s = android.provider.Settings.Secure.getString(mContentResolver,
systemSettingKey);

        if (s == null)
            return "";

        return s;
    }

    private void setString(String systemSettingKey, String value) {
        android.provider.Settings.Secure.putString(mContentResolver, systemSettingKey,
value);
    }

    public boolean isSecure() {
        long mode = getKeyguardStoredPasswordQuality();
        final boolean isPattern = mode == DevicePolicyManager.PASSWORD_QUALITY_SOMETHING;
        final boolean isPassword = mode == DevicePolicyManager.PASSWORD_QUALITY_NUMERIC
                || mode == DevicePolicyManager.PASSWORD_QUALITY_ALPHABETIC
                || mode == DevicePolicyManager.PASSWORD_QUALITY_ALPHANUMERIC;
        final boolean secure = isPattern && isLockPatternEnabled() &&
savedPatternExists()
                || isPassword && savedPasswordExists();
        return secure;
    }

    /**
     * Sets the text on the emergency button to indicate what action will be taken.
     * If there's currently a call in progress, the button will take them to the call
     * @param button the button to update
     */
    public void updateEmergencyCallButtonState(Button button) {
        int newState = TelephonyManager.getDefault().getCallState();
        int textId;
        if (newState == TelephonyManager.CALL_STATE_OFFHOOK) {
            // show "return to call" text and show phone icon
            textId = R.string.lockscreen_return_to_call;
            int phoneCallIcon = R.drawable.stat_sys_phone_call;
            button.setCompoundDrawablesWithIntrinsicBounds(phoneCallIcon, 0, 0, 0);
        } else {
            textId = R.string.lockscreen_emergency_call;
            int emergencyIcon = R.drawable.ic_emergency;
            button.setCompoundDrawablesWithIntrinsicBounds(emergencyIcon, 0, 0, 0);
        }
        button.setText(textId);
    }

    /**
     * Resumes a call in progress. Typically launched from the EmergencyCall button
```

```
     * on various lockscreens.
     *
     * @return true if we were able to tell InCallScreen to show.
     */
    public boolean resumeCall() {
        ITelephony phone =
ITelephony.Stub.asInterface(ServiceManager.checkService("phone"));
        try {
            if (phone != null && phone.showCallScreen()) {
                return true;
            }
        } catch (RemoteException e) {
            // What can we do?
        }
        return false;
    }
}
```

Appendix 4 – Android Hash Rainbow Table Generation (Andersson, 2011)

```
<?php
/**
 * @author Rickard Andersson <h05rikan@du.se>
 * @package AndroidLockScreen
 *
 * This script will generate every combination of the android lockscreen gesture,
calculate
 * it's corresponding hash and save it to a database.
 *
 * This code is released without any license whatsoever, you're free to do what you want
with it.
 */

$dsn = "mysql:dbname=AndroidLockScreen;host=127.0.0.1"; // Change this if you want to
$user = ""; // Change this
$pass = ""; // Change this

try {
    $dbh = new PDO($dsn, $user, $pass);
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
    die();
}

$sth = $dbh->prepare('INSERT INTO RainbowTable (combination, hash) VALUES (?, ?)');

$total = 0;

// Generating all the combinations of gestures with length three, four, ..., eight and
nine
for ($x = 3; $x <= 9; $x++) {
  echo "==> Generating table for $x ... \n";

  $p = new Permutations(9, $x);

  $combinations = 0;

  $values = $p->getCurrent();

  do {
    $str = "";

    foreach ($values as $value) {
      $str .= chr($value);
    }

    $hash = sha1($str);

    if ($sth->execute( array( implode(",", $values) , $hash) )) {
      $combinations++;
    } else {
      echo "Error: ";
      var_dump($sth->errorInfo());
    }

  } while (is_array($values = $p->getNext()));

  $total += $combinations;

  echo "==> Done! Inserted $combinations records!\n";
}

echo "==> All done with a total of $total records!\n";

/**
 * Class for generating permutations of numbers
 * @author Rickard Andersson
 */
class Permutations {
```

```
  private $n; // How many numbers in each permutation to generate
  private $maxValue; // The maximum value in each number
  private $currentValues; // Array of current values

  /**
   * Public constructor to initiate the class with number boundaries
   * @param int $max  The highest number in this series
   * @param int $n    How many numbers to choose from the series of numbers
   */
  public function __construct($max, $n) {
    $this->n = $n;
    $this->max = $max;
    $this->currentValues = range(0, $n -1);
  }

  /**
   * Get the current set of values
   * @return array
   */
  public function getCurrent() {
    return $this->currentValues;
  }

  /**
   * Increase the number and return the resulting set of values
   * @return array|boolean   Returns false when all permutations have been generated.
   */
  public function getNext() {

    if ( $this->increase($this->n - 1) ) {
      return $this->currentValues;
    } else {
      return false;
    }
  }

  /**
   * Increases the value of the number in the index $index in the set of numbers and
checks that
   * the set of numbers still is unique and within boundaries.
   * @param int $index  Which index to increas
   * @return int  Returns the new value on success and -1 on failure
   */
  private function findNext($index) {

    $newValue = $this->currentValues[ $index ];

    do {
      $newValue++;
    } while (in_array($newValue, $this->currentValues));

    if ($newValue >= $this->max) {
      return -1;
    } else {
      return $newValue;
    }
  }

  /**
   * Increases the value of the complete set of values until every value has been
generated. This is a recursive function and a
   * call to $index - 1 will be done if the value of the current index gets above the
given boundary.
   * @param int $index
   * @return bool
   */
  private function increase($index) {

    $this->currentValues[ $index ] = $this->findNext($index);

    // findNext() returns -1 on failure and if $index == 0 all the numbers has been
generated
```

```
    if ($this->currentValues[ $index ] == -1) {
      if ($index == 0) {
        return false;
      }

      // There might still be numbers to generate, call increase() and try to find a new
set of numbers
      // by increasing the number of the index right below this one.
      else {
        $success = $this->increase( $index - 1 );

        // Found a new set of values to work with. Since findNext has returned -1 in the
first call, calling findNext again
        // from -1 and upwards will get the lowest available number for this index.
        if ($success === true) {
          $this->currentValues[ $index ] = $this->findNext($index);
          return true;
        } else {
          return false;
        }
      }
    } else {
      return true;
    }
  }
}
```